The Executor Concept Hierarchy Needs a Single Root

Document #:	P1738R0
Date:	2019-06-17
Project:	Programming Language C++ SG1 LEWG
Reply-to:	Eric Niebler <eniebler@fb.com>

1 Abstract

The executor concepts in [P0443R10] (`OneWayExecutor` and `BulkOneWayExecutor`) do not participate in a subsumption hierarchy with a root `Executor` concept that can be used to constrain an algorithm that schedules work. The design of P0443 is intended to accommodate even more executor concepts – also not in any subsumption relationship to a conceptual root – over time. In the place of a usable executor concept hierarchy, P0443 has a mechanism for querying an executor whether it supports a particular interface and fetching that interface, similar to locale facets, but statically polymorphic.

This design has negative consequences for generic components that accept executors. In particular, it has disastrous consequences for overloading that become worse over time as third parties add more executor concepts.

2 Without a usable root Executor concept

2.1 Writing a generic async algorithm, take 1

[P0443R10] specifies concepts in the pre-C++20 style using requirements tables. Both of its executor concepts refer to the "General requirements of executors" in section (1.2.2). These requirements are:

— `CopyConstructible`
— `EqualityComparable`
— `is_executor_v<Ex> == true`

Additionally, the operations are required to be non-throwing and free of data races.

We can affix a label to this collection of requirements and call it `Executor`. So, does P0443 have a hierarchy with a single root? To answer that, let's try to write a generic algorithm that is constrained with `Executor`.

```
namespace std
{
  // Constrain the executor with Executor, the root of the Executor concept hierarchy(?):
  template<Executor Exec, RandomAccessIterator I>
  void simple_sort( Exec ex, I first, I last )
  {
    if constexpr ( can_require_concept_v< Exec, bulk_oneway_t > )
    {
      auto ex2 = require_concept( bulk_oneway, ex );
      // Bulk oneway implementation in terms of ex2
    }
    else if constexpr ( can_require_concept_v< Exec, oneway_t > )
    {
      auto ex2 = require_concept( oneway, ex );
      // Oneway implementation in terms of ex2
    }
    else
    {
      // OOPS, what do we do here?
    }
  }
}
```

As shown above, if we constrain the algorithm with the `Executor` concept, we end up stuck because there is no interface that is *required* for particular model of `Executor` to support. For instance, a user might pass to this function an executor that implements a `third_party_whizbang_t` executor interface without also providing a conversion to either the `oneway_t` or `bulk_oneway_t` interfaces, which would result in a hard error.

In other words, P0443 does *not* have a usable Executor concept that can function as the root of a hierarchy.

2.2 Writing a generic async algorithm, take 2

Constraining a generic algorithm with P0443's (implicit) `Executor` concept is insufficient. Let's additionally require that the executor type provides either the `oneway_t` or the `bulk_oneway_t` interface.

```
namespace std
{
  // Additionally constrain the executor so that it provides either the oneway
  // or the bulk oneway interface:
  template<Executor Exec, RandomAccessIterator I>
    requires can_require_concept_v< Exec, bulk_oneway_t > ||
      can_require_concept_v< Exec, oneway_t >
  void simple_sort( Exec ex, I first, I last )
  {
    if constexpr ( can_require_concept_v< Exec, bulk_oneway_t > )
    {
      auto ex2 = require_concept( bulk_oneway, ex );
      // Bulk oneway implementation in terms of ex2
    }
    else
    {
      auto ex2 = require_concept( oneway, ex );
      // Oneway implementation in terms of ex2
    }
  }
}
```

This works now, at the cost of some verbosity in the algorithm constraints.

If we consider the stated intention of P0443 to have a user-extensible Executor concept hierarchy with additional interface-changing executor properties provided by third parties, we run into additional problems.

2.3 Function overloading in the presence of interface-changing properties

Suppose the Acme Parallel Runtime vendor devises a custom mechanism for launching work in bulk that is better that `bulk_oneway_t`, so they implement an interface-changing property `acme_bulk_oneway_t` that can be used to accelerate algorithms. They also use `acme_bulk_oneway_t` to provide an accelerated `simple_sort` algorithm in their namespace that outperforms the default `simple_sort` algorithm, which knows nothing about `acme_bulk_oneway_t`.

```
namespace acme
{
  // Constrain the executor with can_require_concept_v< acme_bulk_oneway_t >
  template<std::Executor Exec, std::RandomAccessIterator I>
    requires std::can_require_concept_v< Exec, acme_bulk_oneway_t >
  void simple_sort( Exec ex, I first, I last )
  {
    auto ex2 = std::require_concept( acme_bulk_oneway, ex );
   // An implementation of simple_sort in terms of acme_bulk_oneway
  }
}
```

This works well until someone tries to call `simple_sort` with an executor that implements both `acme_bulk_oneway_t` and either `oneway_t` or `bulk_oneway_t`. Then the call to `simple_sort` is ambiguous because neither the overload in namespace `std` nor the one in namespace `acme` have constraints that subsume the other.

2.4 Disjunction in the Executor concept

Imagine that instead of the degenerate `Executor` concept defined above, we defined the `Executor` concept to be the disjunction between `OneWayExecutor` and `BulkOneWayExecutor`. Does that solve any problems?

```
namespace std
{
  template <class Ex>
  concept Executor =
    CopyConstructible<Ex> &&
    EqualityComparable<Ex> &&
    is_executor_v<Ex> &&
    (can_require_concept_v< Exec, bulk_oneway_t > ||
      can_require_concept_v< Exec, oneway_t >);
}
```

We can now constrain algorithms to take an `Executor` without verbose `requires` clauses, and users can pass types that satisfy one or both of the standard executor concepts.

```
namespace std
{
  // Require users to pass something that satisfies one of standard
  // executor concepts
  template< Executor Exec, RandomAccessIterator I >
  void simple_sort( Exec ex, I first, I last )
  {
    if constexpr ( can_require_concept_v< Exec, bulk_oneway_t > )
    {
      auto ex2 = require_concept( bulk_oneway, ex );
      // Bulk oneway implementation in terms of ex2
    }
    else
    {
      auto ex2 = require_concept( oneway, ex );
      // Oneway implementation in terms of ex2
    }
  }
}
```

Disregarding the fact that disjunctions in concept hierarchies cause worst-case exponential behavior in partial ordering by constraints ([temp.constr.order]), this formulation of the `Executor` concept has problems. The implications are:

— Every algorithm constrained with this `Executor` concept is required to provide two implementations: one for one-way executors, and another for bulk one-way executors.

— We would never be able to extend the `Executor` concept to include other types of execution interfaces that we might add in the future. That would force everybody who wrote an algorithm constrained with this concept to provide an additional implementation expressed in terms of that new interface. Recall that enabling open extension of the Executor concept design space is one of the stated intentions of the design of P0443's properties mechanism.

This also does not solve the problem with ambiguous overloading.

In short, we can have either an open, extensible, *ad hoc* collection of Executor concepts; or maintainable generic async algorithms and open overloading, but not both.

3 With a usable root Executor concept

Instead of the P0443 model, where all *usable* executor concepts are siblings, let's take one concept – say, `OneWayExecute` – rename it "`Executor`", and promote it to be the root of an executor hierarchy. Note that in so doing, we require all executor authors to provide a usable one-way execution interface, even if that is not interesting for them. Additionally note that in moving one concept to the root of a hierarchy, we are not requiring that such a hierarchy be *linear*. There may be many sibling concepts under the root `Executor` concept, just as directly under C++20's `Range` concept there is `View`, `SizedRange`, `InputRange`, and `OutputRange`.

Here is what an `Executor` concept might look like:

```
template <class Ex, class Fn = void(*)()>
concept Executor =
  Invocable<Fn> &&
  CopyConstructible<Ex> &&
  EqualityComparable<Ex> &&
  requires (Ex&& ex, Fn&& fn) {
    ((Ex&&) ex).execute((Fn&&) fn);
  };
```

We could add refinements of this concept, like `BulkExecutor`:

```
template <class Ex, [...additional args]>
concept BulkExecutor =
  Executor<Ex> && [...additional requirements];
```

With an `Execute` concept at the root of the hierarchy, we can express the `simple_sort` example as follows:

```
namespace std
{
  // Constrain simple_sort to simply take an Executor:
  template< Executor Exec, RandomAccessIterator I>
  void simple_sort( Exec ex, I first, I last )
  {
    if constexpr ( BulkExecutor< Exec > )
    {
      // Bulk oneway implementation
    }
    else
    {
      // Oneway implementation
    }
  }
}
```

Third party code would be able to add refinements of any of the standard executor concepts, as follows:

```
namespace acme
{
  template <class Ex>
  concept AcmeBulkExecutor =
    std::Executor<Ex> &&
    [...additional requirements]
}
```

Given the `AcmeBulkExecutor` concept, users would be free to add an overload constrained with that concept without introducing an ambiguity into the overload set:

```
namespace acme
{
  // Constrain the algorithm with AcmeBulkExecutor
  template< AcmeBulkExecutor Exec, std::RandomAccessIterator I>
  void simple_sort( Exec ex, I first, I last )
  {
    // ... Acme bulk implementation
  }
}
```

This introduces no ambiguity because `AcmeBulkExecutor` subsumes `Executor`, so the compiler can order the overloads.

If any down-casting of executors must take place, it is the responsibility of the *callers* of the `simple_sort` algorithm, not the `simple_sort` algorithm itself. That is how to make the set of executor concepts extensible while also guarding against ambiguity and keeping libraries of async algorithms maintainable over time.

4 Satisfying multiple Executor concepts

With an Executor hierarchy that has a single root, there is nothing wrong with an executor satisfying multiple concepts simultaneously; e.g., a `std::BulkExecutor` and an `acme::AcmeBulkExecutor`, just like `std::span` is simultaneously a `ContiguousRange`, a `View`, and a `SizedRange`. There is *still* no ambiguity in the call to `simple_sort`.

There are a number of implementation strategies an executor might use to satisfy multiple executor concepts:

— It may natively provide multiple interfaces on top of its execution context, or
— It may use something like `require_concept` internally to create the necessary executor on demand, or (if that is too expensive)
— It may be implemented as a `std::variant` of a handful of concrete executors created on demand and cached.

An algorithm that needs both a Standard and an Acme bulk executor would simply require both:

```
template < std::BulkExecutor Ex >
  requires acme::AcmeBulkExecutor<Ex>
void the_best_algorithm( Ex ex, ... );
```

It is up to the caller of `the_best_algorithm` to provide an executor that satisfies both concepts (possibly with one of the approaches described above).

5 Conclusion

Although intended to permit a natural evolution within the executor design space over time, the *ad hoc* and fungible executor concept hierarchy implicit in the design of P0443 is actively hostile to generic programming and gets in the way of the very kind of design evolution it was intended to enable. We recommend a executor hierarchy with a single root to avoid intractable maintenance problems.

6 References

[P0443R10] Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, H. Carter Edwards, Gordon Brown, David Hollman. 2019. A Unified Executors Proposal for C++.
https://wg21.link/p0443r10